Capítulo 1

Programación estructurada

1.1. Tipos de datos

El lenguaje Python ofrece una importante variedad de tipos de datos, de entre los cuales se destacan los siguientes:

Categoría	Tipo	Nombre	Uso
Texto	Cadena	str	Cadenas de caracteres de longitud arbitraria
Numérico	Entero	int	Números enteros
Numérico	Flotante	float	Números con punto flotante
Numérico	Complejo	complex	Números complejos
Lógico	Booleano	bool	Valores de verdad
Secuencia	Tupla	tuple	Secuencia inmutable de valores
Secuencia	Lista	lista	Secuencia mutable de valores
Secuencia	Rango	range	Secuencia inmutable de números enteros
Conjunto	Conjunto	set	Conjunto de valores sin repetición y con búsquedas rápidas
Asociación	Diccionario	dict	Pares ordenados clave / valor
Vacío	Sin tipo	none	Variables declaradas pero sin valor ni tipo

1.2. Operadores

Los datos de cada uno de los tipos del lenguaje pueden ser manipulados mediante un conjunto de operadores. Un operador es un símbolo o una palabra que realiza una operación sobre uno o más valores. Los valores pueden ser constantes, variables o expresiones. Los operadores pueden manipular una cantidad diferente de valores. La mayoría de los operadores se denominan binarios porque requieren dos operandos que es indican antes y después del operador. Asimismo existen algunos operadores unarios y un único operador ternario.

1.2.1. Operadores aritméticos

Requieren dos operandos numéricos de cualquier tipo. Si ambos operandos son del mismo tipo, es decir, dos enteros o dos flotantes el resultado generalmente será del mismo tipo de los operandos. En caso de que sean de tipos diferentes, el resultado será del tipo más grande. Por lo tanto, si se opera con un entero y un flotante, el resultado será flotante.

Símbolo	Uso	Observaciones
+	Suma	Aplicado a cadenas efectúa una concatenación
-	Diferencia	
*	Producto	Si un operando es una cadena y el otro es un entero, repite la cadena tantas veces como indique el operando numérico
/	Cociente	Siempre retorna un float
//	Cociente entero	Siempre retorna un int truncando los decimales
%	Módulo	Retorna el resto de una división
**	Potencia	Retorna el primer operando elevado a la potencia indicada por el segundo.

1.2.2. Operadores de asignación

Los operadores de asignación siempre son binarios, de los cuales el primer operando debe ser una variable o cualquier lugar donde se pueda guardar un valor, tal como un parámetro actual y el segundo debe ser una expresión. El tipo retornado siempre es el de la expresión asignada.

Símbolo	Uso	Observaciones
=	Asignación	
+=	Acumulación o incremento	El primer operando debe ser una variable a la cual se le incrementa su valor con el resultado de la expresión del segundo operando
-=	Decremento	
*= /=	Combinados	Todas las operaciones aritméticas pueden ser combinadas con asignación, aunque su uso es infrecuente

1.2.3. Operadores de comparación

Los operadores de comparación comparan los valores de dos operandos de un mismo tipo y retornan un valor de verdad. Si los tipos comparados son diferentes, se promueve uno de esos al tipo más amplio.

Símbolo	Uso	Observaciones
==	Igualdad	
!=	Diferencia	
>	Mayor que	
<	Menor que	
>=	Mayor o igual	
<=	Menor o igual	

Existe un operador de comparación cuya sintaxis y comportamiento son diferentes al resto. Es el único operador ternario del lenguaje y permite escribir como una expresión un cálculo que de otra forma requiere una instrucción condicional `if-else`. La sintaxis del operador es:

```
[valor si verdadero] if [condicion] else [valor si falso]
```

El operador evalua la condición, la cual debe evaluarse o convertirse a un valor de verdad, si la misma es verdadera se retorna el primer operando y en caso contrario el último.

```
importe = 45345
saldo = "Acreedor" if importe > 0 else "Deudor"
# La variable saldo finaliza con valor "Acreedor"

existe = False
print("Si" if existe else "No")
# Imprime "Si"
```

1.2.4. Operadores lógicos

Permiten efectuar operaciones de algebra booleana entre dos valores de verdad. Si los operandos no son booleanos los convierte evaluando los valores vacíos de cada tipo de datos como falsos, y cualquier otro valor como verdadero.

Símbolo	Uso	Observaciones
and	Y lógico	Aplica corto circuito, si el primer operando es falso, retorna falso sin evaluar el segundo
or	O lógico	Aplica corto circuito, si el primer operando es verdadero, retorna verdadero sin evaluar el segundo
not	Negación	Es un operador unario, invierte el valor de verdad indicado a continuación del mismo

1.3. Estructuras condicionales

1.3.1. Instrucción if

La instrucción if en Python permite ejecutar un bloque de código si se cumple una condición. La sintaxis general es:

```
if condicion:
    bloque de codigo
```

La condición puede ser una expresión lógica o booleana que se evalúa como verdadera (True) o falsa (False). Si la condición es verdadera, se ejecuta el bloque de código indentado después de los dos puntos (:). Si la condición es falsa, se salta el bloque de código y se continúa con el resto del programa.

Por ejemplo:

```
if x > 0:
    print("x es positivo")
```

En este caso, se imprime el mensaje ”x es positivo”solo si la variable x tiene un valor mayor que cero. De lo contrario, no se hace nada.

La instrucción if se puede combinar con else o elif para definir otros casos alternativos. Else se usa para ejecutar un bloque de código si la condición del if no se cumple. Elif se usa para evaluar otra condición después del if. Se pueden usar varios elif para crear múltiples ramas.

Por ejemplo:

```
if x > 0:
    print("x es positivo")
elif x < 0:
    print("x es negativo")
else:
    print("x es cero")
```

En este caso, se imprime un mensaje diferente según el valor de x. Si x es positivo, se imprime “x es positivo”. Si x es negativo, se imprime “x es negativo”. Si x es cero, se imprime “x es cero”.

1.3.2. Condiciones

Las instrucciones y cláusulas que evalúan condiciones exigen que la condición sea escrita como una expresión booleana. Como tal, una expresión puede ser una constante, una variable o el resultado de una operación.

Aunque es muy infrecuente, puede plantearse una condición con un valor True constante, pero esta variante tiene sentido únicamente si se plantea un ciclo infinito.

En cambio sí es muy habitual evaluar variables de tipo boolean, conocidas generalmente como banderas o centinelas. Para estas variables, la condición se redacta únicamente con el nombre de la variable. Durante la ejecución, el valor que la misma almacene servirá para ejecutar o no el bloque de la instrucción condicional. De esa manera si se dispone de una variable llamada `existe`, la redacción preferida de una instrucción if sera `if existe:`, siendo innecesario y hasta negativo el uso de una comparación con `if existe == True:`. De la misma manera, si se requiere evaluar que la variable contenga un valor falso, la redacción preferida es `if not existe:` en lugar de comprarar `if existe == False:`.

En las condiciones que involucren variables de tipos diferentes al boolean, la expresión a evaluar se redacta como una operación que utilice los operadores que devuelven valores de verdad, es decir, los operadores de comparación y los operadores lógicos.

1.3.3. Conversiones a boolean

Python ofrece un mecanismo de conversiones implícitas de todos los tipos de datos a boolean. Esta caracterísitica permite evaluar como verdaderas o falsas variables o expresiones de otros tipos aplicando ciertas reglas de conversión.

Esta característica se denomina *truthiness* y puede ser fuente de serios errores hasta que se la comprende acabadamente. La idea principal es que todo valor de un tipo diferente al boolean puede ser interpretado “como si fuera verdadero” (*truthy*) o “como si fuera falso” (*falsy*).

El criterio general es que se considera falsy todo dato “vacío”, dependiendo el valor exacto del tipo de datos. En el caso de los datos más simples el vacío es el más natural: 0 para los tipos numéricos, cadena vacía en el caso de las cadenas y None en el caso de las referencias a estructuras u objetos. Por otro lado, en los casos de las secuencias y estructuras de datos, se evalúan como falsy si están vacías, es decir, sin elementos.

Esta conversión implícita puede ser utilizada en ciertas condiciones para mejorar la legibilidad de la misma. De la misma manera en que para evaluar una bandera es preferible evitar la comparación con los valores True y False, se puede aprovechar la misma idea al evaluar si una lista está vacía o si una variable numérica es 0.

Con este criterio, las siguientes condiciones son válidas:

```
nombre = input("Ingrese su nombre")
edad = int(input("Ingrese su edad))

if not nombre: print("No ingresó su nombre")
if not edad: print("Debe ingresar una edad diferente a 0")

print("Su edad es " + ("impar" if edad % 2 else "par"))
```

1.3.4. Combinación de condiciones

Una característica particular del lenguaje python que siendo muy útil resulta difícil de aprender es la posibilidad que brinda para combinar operadores condicionales.

Una situación muy habitual es la de verificar si un valor numérico se encuentra en un rango. Para ello la solución más simple es la de comparar el valor con cada uno de los extremos del rango y conectar con el operador AND:

```
if nota > 0 and nota ≤ 10:
```

Pero dado que las condiciones que evalúan cada extremo del rango involucran a la misma variable y que ambas están conectadas por and, se pueden combinar de la siguiente forma:

```
if 0 < nota ≤ 10:
```

Debe prestarse especial atención a que esta posiblidad puede llevar a errores difíciles de detectar. Por ejemplo, es una situación razonable determinar que dos variables cumplan con la misma condición, por caso, que sean mayores a 5. La única forma adecuada es la de evaluar `if a > 5 and b > 5`. Pero es tentador intentar una combinación de la forma:

```
if a and b > 5:
```

La condición anterior, aunque es válida desde el punto de vista de la sintaxis del lenguaje, no evalúa lo que parece a simple vista: el operador and posee como primer operando una variable numérica la cual evalúa como truthy si es distinta de 0, por lo tanto dicha condición es equivalente a `if a ≠ 0 and b > 5`.

Otro inconveniente similar ocurre si se intenta combinar las condiciones como:

```
if a > b > 5:
```

En este otro caso ocurre algo similar, la sintaxis es correcta, pero el funcionamento es algo diferente. Esta condición efectivamente verifica que ambas variables sean mayores a 5, pero además exige que la segunda sea mayor a la primera, restricción que no estaba solicitada originalmente.

1.4. Estructuras repetitivas

1.4.1. Instrucción for

La instrucción for permite iterar sobre una secuencia de elementos, como un rango, una lista, una tupla o un diccionario. La sintaxis general de la instrucción for es la siguiente:

```
for elemento in secuencia:
    # hacer algo con el elemento
```

La variable elemento toma el valor de cada elemento de la secuencia en cada iteración del bucle. El bloque de código que se ejecuta en cada iteración debe estar indentado. El bucle termina cuando se recorre toda la secuencia o cuando se encuentra una instrucción break.

La instrucción for es utilizada para realizar tareas repetitivas con los elementos de una secuencia, como sumarlos, modificarlos o filtrarlos. También se puede usar para crear nuevas secuencias a partir de otras existentes, usando la comprensión de listas, tuplas o diccionarios.

Por ejemplo, para recorrer una lista se puede escribir el siguiente ciclo:

```
frutas = ["manzana", "banana", "naranja"]

# Por cada fruta de la lista de frutas:
for fruta in frutas:
    print(fruta)
```

1.4.2. Instrucción while

La instrucción “while” en Python se utiliza para crear un bucle o ciclo que se repite mientras se cumple una condición específica. A diferencia de la instrucción “for” que se utiliza para iterar sobre una secuencia conocida, la instrucción “while” se repite hasta que una condición se evalúa como falsa. La sintaxis general de la instrucción es la siguiente:

```
while condición:
   # bloque iterativo
```

En el ejemplo anterior “condición” es una expresión que se evalúa en cada iteración del bucle. Si la condición es verdadera, el bloque de código dentro del bucle se ejecuta. Si la condición es falsa el ciclo finaliza.

Es importante tener cuidado al utilizar la instrucción while para evitar que el bucle se convierta en un ciclo infinito. Para evitar esto, es común utilizar una lógica dentro del bucle que modifique las variables involucradas en la condición para que eventualmente se evalúe como falsa y el ciclo se detenga.

En el siguiente ejemplo se obtiene la suma de todos los dígitos de un número con operaciones aritméticas, por medio de la extracción del último dígito con el operador de módulo. En cada vuelta se desplazan los restantes digitos efectuando una división entera en 10. Dado que no necesariamente se conoce la cantidad de dígitos del número ingresado, se debe utilizar un ciclo while que dará una vuelta por cada dígito, y finaliza cuando el número se reduce a 0.

```
numero = int(input("Ingrese un número: "))
suma_digitos = 0

while numero > 0:
    # Obtiene el último dígito del número
    digito = numero % 10

    # Agrega el dígito a la suma total
    suma_digitos += digito

    # Elimina el último dígito del número
    numero //= 10

print("La suma de los dígitos es:", suma_digitos)
```

1.4.3. Saltos

La instrucción **break** se utiliza para terminar un bucle antes de que se complete su condición de finalización. Debe encontrarse dentro de alguna instrucción condicional que determine el momento en que debe interrumpirse el ciclo. La instrucción break solo afecta al bucle en el que se encuentra y no a los bucles anidados o externos.

La instrucción **continue** se utiliza dentro de ciclos para omitir el resto del código en una iteración actual y pasar

a la siguiente iteración. Cuando se encuentra una instrucción continue, el flujo de ejecución del programa salta inmediatamente al principio del ciclo, sin ejecutar el código restante de esa iteración en particular.

1.4.4. Cláusula else

La cláusula `else` en los ciclos se utiliza para especificar un bloque de código que se ejecuta cuando el ciclo ha terminado de iterar sobre todos los elementos de una secuencia o cuando la condición del ciclo se evalúa como falsa. La cláusula else se ejecuta después de que el ciclo ha finalizado de forma normal, es decir, sin interrupciones como break o return.

En el siguiente ejemplo se verifica la existencia de la letra A en una cadena. En el momento en que se encuentra la primera aparición se interrumpe el ciclo con break para no continuar con el recorrido. Por lo tanto, que el ciclo finalice normalmente, sin la interrupción, significa que la letra no fue encontrada:

```
texto = input("Ingrese un texto: ")

for letra in texto:
    if letra == "A":
        print("Contiene la letra A")
        break
else:
    print("No contiene la letra A")
```

1.5. Funciones

1.5.1. Sintaxis

Las funciones se definen utilizando la palabra clave `def` seguida del nombre de la función. Luego del nombre deben existir un par de paréntesis que pueden contener los parámetros de la función y el símbolo de dos puntos (:). A continuación, se escribe el bloque de código de la función indentado.

```
def nombre_de_funcion(parametro1, parametro2, ...):
    # Bloque de código de la función
    # Puede incluir declaraciones, operaciones y retornos
```

En el siguiente ejemplo se define una función con retornos y parámetros:

```
def calcular_area_triangulo(base, altura):
   area = (base * altura) / 2
   return area
```

La función llamada `calcular_area_triangulo` que toma dos parámetros base y altura. Dentro del bloque de código de la función, se calcula el área de un triángulo utilizando la fórmula correspondiente y se almacena en la variable area. Luego, se utiliza la palabra clave `return` para devolver el valor calculado de area como resultado de la función.

Después de definir una función, puede ser invocada desde otro lugar del código utilizando su nombre y proporcionando los argumentos necesarios. De esta manera la función anterior puede ser invocada como:

```
resultado = calcular_area_triangulo(5, 3)
print("El área del triángulo es:", resultado)
```

En el caso de que una función no incluya una instrucción return o posea una instrucción return sin indicar ningún valor de retorno, la misma retorna automáticamente un valor de None al finalizar su ejecución.

1.5.2. Parámetros

Existen diversas formas de indicar los parámetros de una función, cada una de las cuales tiene una utilidad bien marcada.

Parámetros posicionales: Son los parámetros que se pasan a la función en el mismo orden en el que se definen en la firma de la función. Estos parámetros son obligatorios y deben ser proporcionados al llamar a la función.

```
def saludar(nombre, edad):
    print("Hola", nombre, "tenés", edad, "años.")

saludar("Olga", 25)
# Ejemplo de llamada con parámetros posicionales
```

Parámetros con valor predeterminado: Son parámetros que tienen un valor asignado por defecto en caso de que no se les pase un valor al llamar a la función. Estos parámetros son opcionales.

```
def saludar(nombre, edad=30):
    print("Hola", nombre, "tenés", edad, "años.")

saludar("Jorge")
# Ejemplo de llamada sin proporcionar el parámetro "edad"
```

Parámetros de palabra clave: Se especifican durante la llamada a la función utilizando el formato `nombre_parametro=valor`. Estos parámetros son opcionales, por lo tanto permiten omitir aquellos que posean un valor por defecto. Asimismo permiten especificar los argumentos en cualquier orden.

```
def saludar(nombre, edad):
    print("Hola", nombre, "tenés", edad, "años.")

saludar(edad=40, nombre="Carlos")
# Ejemplo de llamada con parámetros de palabra clave
```

Parámetros variables (*args): Permite pasar un número variable de argumentos posicionales a una función. Los argumentos se agrupan en una tupla dentro de la función.

```
def sumar(*numeros):
   total = 0
   for num in numeros:
      total += num
   return total

resultado = sumar(1, 2, 3, 4, 5)
# Ejemplo de llamada con parámetros variables
```

Parámetros de palabras clave variables (kwargs):** Permite pasar un número variable de argumentos de palabra clave a una función. Los argumentos se agrupan en un diccionario dentro de la función.

```
def imprimir_datos(**datos):
   for clave, valor in datos.items():
      print(clave + ":", valor)

imprimir_datos(nombre="Juana", edad=25, ciudad="Pinamar")
# Ejemplo de llamada con parámetros de palabras clave variables
```

1.5.3. Retorno

Las funciones pueden retornar cero, uno o más expresiones como resultados. En el caso de no incluir ninguna instrucción return, u omitir el valor retornado, la misma entrega un valor de None.

Para que una función retorne más de un valor, se los debe indicar separados por comas. En la invocación a la misma los valores retornados pueden ser asignados a una serie de variables también separadas por comas.

Por ejemplo, la siguiente función recibe como parámetro una lista de números y devuelve la sumatoria de los mismos y su promedio:

```
def promedio_suma(numeros):
    suma = 0

    for x in numeros:
        suma += n
    promedio = suma / len(numeros)

    return promedio, suma

lista = [10, 20, 30, 40, 50]
promedio, suma = promedio_suma(lista)

print("Promedio:", promedio)
print("Suma:", suma)
```

1.6. Ejercicios

1.6.1. Estación de servicio

Una estación de servicio que dispone de 10 surtidores y necesita gestionar información relacionada con la venta de combustibles en la jornada.

De cada surtidor se conoce:

- Número de Surtidor (validar que sea un número entre 1 y 30)
- Cantidad: representa la cantidad de litros de combustible vendido por el surtidor (validar que sea un número positivo)
- Tipo: representa el tipo de combustible del surtidor. Los valores que puede asumir son 1 representa “Nafta Super”, 2 representa “Nafta Especial” y 3 representa “Gasoil” (validar que se ingresen valores válidos).

Se pide calcular e imprimir:

- El total de litros vendidos en la jornada, por tipo de combustible.
- El número de surtidor que menos combustible vendió.
- El promedio por surtidor en litros de combustible vendido en la jornada (promedio general, es un único resultado).

estacion.py

```
total_nafta_super = total_nafta_especial = total_gasoil = 0
menor = None
surtidor_menor = None
total = 0

for i in range(10):
    numero = int(input('Ingrese número de surtidor (1 a 30): '))
    while not 0 < numero ≤ 30:
        print('Ingresó un número inválido')
        numero = int(input('Ingrese número de surtidor (1 a 30): '))

    cantidad = int(input('Ingrese la cantidad de litros vendidos: '))
    while cantidad < 0:
        print('Debe ingresar un número positivo')
        cantidad = int(input('Ingrese la cantidad de litros vendidos: '))

    tipo = int(input('Ingrese el tipo de combustible (1 a 3): '))
    while not 1 ≤ tipo ≤ 3:
        print('Ingresó un número inválido')
        tipo = int(input('Ingrese el tipo de combustible (1 a 3): '))

    if tipo == 1:
        total_nafta_super += cantidad
    elif tipo == 2:
        total_nafta_especial += cantidad
    else:
        total_gasoil += cantidad

    if not menor or cantidad < menor:
        menor = cantidad
        surtidor_menor = numero

total = total_nafta_super + total_nafta_especial + total_gasoil
promedio = total // 10

print(f"Total de litros de nafta super: {total_nafta_super}")
print(f"Total de litros de nafta especial: {total_nafta_especial}")
print(f"Total de litros de gasoil: {total_gasoil}")
print(f"Surtidor que menos litros vendió: {surtidor_menor}")
print(f"Promedio de litros por surtidor: {promedio}")
```

1.6.2. Procesamiento de temperaturas en una lista

Ingresar un conjunto de temperaturas en una lista, finalizar la carga cuando se reciba un 50. Sólo aceptar temperaturas entre -20 y 49 grados.

Calcular y mostrar:

- Cantidad de días con temperatura bajo cero
- Promedio de temperaturas
- Promedio de temperaturas de los días cálidos, es decir con temp. mayor a 20
- Mostrar “Si” o “No” para indicar si hubo algún día con más de 40 grados.
- La mayor temperatura de los días que no fueron cálidos
- Cantidad de días con temperatura menor al promedio

validacion.py

```python
def ingresar_numero_entre(mensaje, minimo, maximo):

    valor = float(input(mensaje))
    while not minimo ≤ valor ≤ maximo:
        print(f"Debe ingresar un valor entre {minimo} y {maximo}")
        valor = float(input(mensaje))

    return valor
```

procesolistas.py

```python
def cantidad_menor(lista, techo):
    """
    Cuenta los elementos de una lista que sean menores a un tope

    :param lista: lista de valores
    :type lista: lista de datos que soporte comparación por menor
    :param techo: valor máximo para filtrar
    :type techo: el mismo de los elementos de la lista
    :returns: la cantidad de elementos cuyo valor sea menor al techo
    :rtype: entero
    """
    c = 0
    for x in lista:
        if x < techo:
            c += 1

    return c


def promedio(lista):
    """
    Calcula el promedio simple de todos los elemento de una lista

    :param lista: lista
    :type lista: lista de números
    :returns: el promedio de todos los valores de la lista o 0 si la lista está vacía
    :rtype: flotante
    """
    cantidad = len(lista)
    if cantidad == 0:
        return 0

    suma = 0
    for x in lista:
        suma += x

    return suma / cantidad


def existe(lista, buscado):
    """
    Busca un valor en una lista e informa si lo pudo encontrar

    :param lista: una lista de cualquier tipo
    :param buscado: el valor que se busca
    :tipo buscado: el mismo tipo que el de los datos almacenados en la lista

    :returns: verdadero si el elemento buscado existe en la lista y falso en caso contrario
    :rtype: boolean
    """
    for x in lista:
        if x == buscado:
            return True

    return False
```

temperaturas.py

```
from validacion import *
from proceso_listas import *

def cargar_temperaturas():

    temperaturas = []

    print("Ingrese las temperaturas entre -20 y 49. Finaliza con 50.")
    t = ingresar_numero_entre("Ingrese una temperatura: ", -20, 50)
    while t ≠ 50:
        temperaturas.append(t)
        t = ingresar_numero_entre("Ingrese una temperatura: ", -20, 50)

    return temperaturas


def promedio_mayores(temperaturas, piso):

    cantidad = 0
    suma = 0

    for x in temperaturas:
        if x > piso:
            suma += x
            cantidad += 1

    if cantidad == 0:
        promedio = 0
    else:
        promedio = suma / cantidad
    #promedio = suma / cantidad if cantidad ≠ 0 else 0

    return promedio


def existe_mayor(temperaturas, piso):

    for x in temperaturas:
        if x > piso:
            return True

    return False



def mayor_dias_calidos(temperaturas):

    mayor = None

    for x in temperaturas:
        if x ≤ 20:
            if mayor is None or x > mayor:
                mayor = x

    return mayor


def calcular(temperaturas):

    dias_bajo_cero = cantidad_menor(temperaturas, 0)
    promedio_todos = promedio(temperaturas)
    promedio_dias_calidos = promedio_mayores(temperaturas, 20)
    hubo_40_grados = existe_mayor(temperaturas, 40)
    mayor = mayor_dias_calidos(temperaturas)
    dias_menor_al_promedio = cantidad_menor(temperaturas, promedio_todos)


    print(f"Hubo {dias_bajo_cero} días con temperatura bajo cero")
    print(f"El promedio de temperaturas fue de {promedio_todos}")
    if (promedio_dias_calidos ≠ 0):
        print(f"Y de los días cálidos fue de {promedio_dias_calidos}")

    print("¿Hubo días con más de 40 grados?", "Si" if hubo_40_grados else "No")
    if mayor:
        print(f"La mayor temperatura de los días que no fueron cálidos fue de {mayor}")
```

```
    else:
        print("Todos los días fueron cálidos")

    print(f"Hubo {dias_menor_al_promedio} días con temperatura menor al promedio")


def principal():

    temperaturas = cargar_temperaturas()
    calcular(temperaturas)


if __name__ == "__main__":
    principal()
```

Capítulo 2

Secuencias

2.1. Introducción

Los tipos de datos de secuencias ofrecen la posibilidad de almacenar un conjunto de datos con un único identificador, para poder acceder a los mismos posteriormente en tanto forma individual como grupal. Asimismo, la instrucción for permite recorrer cualquier tipo de secuencia, entregando en cada iteración cada uno de los valores almacenados en la misma.

2.1.1. Tamaño

Se puede obtener el tamaño de una secuencia con la función `len`, pasando la misma como parámetro. La función retorna un valor entero indicando la cantidad de elementos almacenados en la secuencia.

2.1.2. Operador de acceso indexado

Las secuencias disponen del operador de acceso indexado, que se indica como un par de corchetes a continuación del identificador. El mismo permite obtener el valor almacenado en una posición específica de la secuencia, identificando cada una de ellas mediante un número natural que inicia en 0. De esta manera, el primer elemento posee el índice 0, el segundo el 1, etc.

También pueden utilizarse índices negativos para acceder a los elementos desde el último, indicando como -1 al último, -2 al penúltimo y así sucesivamente.

El índice suele redactarse con una constante o variable entera, pero también pueden usarse expresiones.

```
titulo = "Paisaje"

print(titulo[2]) # imprime "i"
print(titulo[-2]) # imprime "j"
print(titulo[len(titulo)//2]) # imprime "s"
```

2.1.3. Operador de rebanadas

El operador de rebanadas o *slicing* también se representa mediante corchetes ([]), pero incluye un rango de índices separados por dos puntos (:). Permite extraer un subconjunto de elementos contiguos de una secuencia. Su retorno siempre será una nueva secuencia que copia los elementos de la original sin modificarla.

El mismo permite indicar un índice inicial y otro final del la sección a rebanar. Si no se indica el inicial, se asume 0; mientras que si no se indica el final, se asume el último. El corte de la secuencia siempre incluye al valor del índice inicial pero no incluye al índice final.

Por ejemplo:

```
titulo = "Paisaje"

print(titulo[1:3]) # imprime "ai"
print(titulo[:3]) # imprime "Pai"
print(titulo[3:]) # imprime "saje"
```

Una variante del operador de rebanadas permite indicar un salto al seleccionar los elementos a cortar. Para ello se puede incluir un segundo símbolo de dos puntos y a continuación un número entero estableciendo que se corten los elementos desde el índice inicial y se salteen los siguientes. Así, con un salto de 2, se extraen las posiciones alternadas, mientras que con un salto de -1 se recorre la secuencia hacia atrás. De esta manera, utilizando una rebanada sin indicar inicio ni final y con salto -1, se obtiene una nueva secuencia inviertiendo la primera.

```
titulo = "Paisaje"

print(titulo[::2]) # imprime "Piae"
print(titulo[5:0:-1]) # imprime "jasia"
print(titulo[::-1]) # imprime "ejasiaP"
print(titulo[::-2]) # imprime "eaiP"
```

2.2. Cadenas

La secuencia más simple es la secuencia de caracteres, tambien denominada cadena de caracteres o simplemente cadena. Las mismas almacenan un conjunto de caracteres como un texto, los cuales pueden ser manipulados en su conjunto o en forma individual.

Para indicar un valor constante se las delimita con comillas simples (') o dobles ("), pero utilizando el mismo símbolo tanto en la apertura como en el cierre.

Las variables de tipo cadena pueden ser asignadas y consultadas como variables para operar con el texto como una unidad, pero tambien puede accederse a cada caracter que conforma la secuencia mediante los operadores de acceso indexado y de rebanadas. El operador de suma (+) une dos cadenas y el operador de multiplicacion repite una cadena una cantidad de veces indicada en el segundo operando.

A continuación se presentan algunas de las operaciones más habituales:

```
mensaje1 = "Hola"
mensaje2 = "mundo!"

# Concatenación
mensaje_concatenado = mensaje1 + " " + mensaje2
print(mensaje_concatenado)  # Resultado: 'Hola mundo!'

# Multiplicacion
print("A" * 10) # Resultado: AAAAAAAAAA

# Obtener una subcadena utilizando rebanada
subcadena = mensaje1[1:3]
print(subcadena)  # Resultado: 'ol'

# Convertir a mayúsculas
mayusculas = mensaje1.upper()
print(mayusculas)  # Resultado: 'HOLA'

# Convertir a minúsculas
minusculas = mensaje2.lower()
print(minusculas)  # Resultado: 'mundo!'
```

2.2.1. Otras operaciones

Nombre	Uso
startswith	Indica si la cadena comienza con una subcadena pasada por parámetro
endswith	Indica si la cadena termina en una subcadena pasada por parámetro
find	Busca una subcadena e informa la posición en que se encuentra
join	Concatena todos los elementos de una secuencia con un delimitador
replace	Reemplaza todas las apariciones de una subcadena por otra
strip	Elimina todos los espacios que se encuentren al inicio y al final
split	Divide la cadena con un delimitador y devuelve una lista con los valores extraidos

2.2.2. Cadenas con formato

Cuando se requiere concatenar valores constantes con valores de variables, la operación de concatenación es impráctica porque no permite especificar fácilmente el formato de cada variable. Frecuentemente se requiere incluir dentro de una cadena larga el valor de una o más variables pero indicando características para la presentación de tales variables como ancho o alineación.

Para ello se dispone de las cadenas formateadas o *f-strings*, las cuales permiten agregar dentro de la cadena de *placeholders* o zonas de reemplazo, indicadas con un par de llaves {}, dentro de cada uno de los cuales se ubica el valor de una variable u operación. Para indicar que una cadena incluye placeholders debe indicarse con una letra f antes de la comilla de apertura.

Por ejemplo, para incorporar el valor de las variables `nombre` y `apellido` dentro de una cadena que contenga un saludo, en lugar de realizar una concatenación se las puede ubicar dentro de la cadena en un placeholder para cada una:

```
saludo = "Hola " + apellido + ", " + nombre + "! ¿cómo estás?"
saludo = f"Hola {apellido}, {nombre}! ¿cómo estás?"
```

A cada valor incrustado se le puede indicar un tamaño mínimo, de forma tal que si el valor no lo cumple se complete con espacios hasta alcanzar el tamaño indicado. Asimismo, se puede especificar una dirección de alineación para que los espacios sean agregados en uno o ambos extremos del valor y que el mismo quede alineado a la izquierda, a la derecha o centrado.

De esta manera, en cada placeholder se puede especificar:

```
f"{expresion:direccion ancho}"
```

La dirección se especifica con los siguientes símbolos

Símbolo	Dirección
>	Alineado a la derecha, agrega espacios antes del valor
<	Alineado a la izquierda, agrega espacios después del valor
^	Centrado, agrega espacios antes y después del valor

En el caso de las expresiones de tipo cadena de caracteres, la alineación por defecto es a la izquierda, mientras que para los valores numéricos la alineación por defecto es a la derecha.

Por ejemplo:

```
print(f"Hola {apellido:>20}, {nombre}, como estás?")
'Hola               Perez, Juan, como estás?'

print(f"Hola {apellido:<20}, {nombre}, como estás?")
'Hola Perez               , Juan, como estás?'

print(f"Hola {apellido:^20}, {nombre}, como estás?")
'Hola        Perez        , Juan, como estás?'
```

La especificación del ancho requiere un número entero, excepto en el caso de la presentación de valores de tipo float. Para esta situación el ancho se indica con un número entero indicando el ancho total, incluyendo la parte entera, el punto de decimal y la parte decimal, seguidos por un punto y la cantidad de dígitos que deben presentarse a la derecha del punto decimal y una letra f. De esta manera, para mostrar una variable con dos decimales se debe especificar {`variable:8.2f`}, y la misma se presenta con 8 caracteres en total, los cuales se reparten con cinco para la parte entera, el punto decimal y dos para los decimales.

2.3. Tuplas

La tupla es un tipo de dato que se utiliza para almacenar una secuencia ordenada e inmutable de elementos. A diferencia de las listas, que se definen utilizando corchetes ([]), las tuplas se definen utilizando paréntesis (()) o simplemente separando los elementos por comas. Por ejemplo:

```
tupla1 = (1, 2, 3)
tupla2 = "a", "b", "c"
```

En estos ejemplos, tanto tupla1 como tupla2 son variables que contienen tuplas.

Las tuplas son similares a las listas en el sentido de que pueden contener múltiples elementos, pero tienen la particularidad de ser inmutables, lo que significa que no se pueden modificar una vez creadas. Esto implica que no se pueden agregar, eliminar o cambiar elementos individualmente en una tupla.

Se pueden acceder a los elementos individuales de una tupla utilizando el operador de acceso indexado. Por ejemplo:

```
tupla3 = (10, 20, 30)
primer_elemento = tupla3[0]  # Acceso al primer elemento de la tupla
print(primer_elemento)  # Resultado: 10
```

En este ejemplo, se utiliza el operador de acceso indexado para acceder al primer elemento de la tupla “tupla3” y se almacena en la variable “primer_elemento”.

Las tuplas son útiles cuando se requiere almacenar una colección de elementos que no deben cambiar, como coordenadas de un punto, información fija o estructuras de datos que no deben modificarse accidentalmente.

Cuando una función retorna una serie de valores separados por comas, en realidad está devolviendo una tupla. Desde la llamada a la misma se puede asignar dicho retorno en una variable que tomará el tipo de datos tupla, o en una serie de variables separadas por comas, de forma tal que cada una de ellas será asignada con cada uno de los valores integrantes de la tupla. Esto se logra mediante una característica del lenguaje denominada desestructuración.

2.3.1. Desestructuración

La desestructuración es una herramienta que ofrece el lenguaje para poder asignar más de una variable a la izquierda del operador de asignación, de forma tal que cada una de tales variables sea asignada con cada uno de los valores de una secuencia que se presente a la derecha del operador.

De esta manera, son válidas las siguientes asignaciones:

```
x, y = 23, 44
# x = 23
# y = 44

precios = [58,22,99]
pre1, pre2, pre3 = precios
# pre1 = 58
# pre2 = 22
# pre3 = 99

letra1, letra2, letra3 = "DAO"
# letra1 = D
# letra2 = A
# letra3 = O
```

Este mecanismo permite que si una función retorna más de un valor los mismos puedan ser asignados en variables individuales al retornar la misma. También permite realizar operaciones que de otra forma requeririan varias operaciones de asignación o incluso variables auxiliares. Es notable el caso del intercambio de dos variables, que puede resolverse de una manera muy simple mediante `a, b = b, a`.

La cantidad de variables asignadas mediante desestructuración debe coincidir con el tamaño de la secuencia. En el caso de que la secuencia asignada tenga una cantidad grande o variable de elementos la desestructuración puede realizarse dejando que alguna de las variables sea indicada con un `*`. De esta manera, en las otras variables se asignan valores individualmente mientras que en la indicada con el asterisco se guarda una nueva secuencia con los valores restantes:

```
persona = "Juan", "Perez", 34, "San Martin 2423", "5000"
nombre, apellido, *otros_datos = persona
# nombre = "Juan"
```

```
# apellido = "Perez"
# otros\_datos =  34, "San Martin 2423", "5000"
```

2.4. Listas

Una lista es un tipo de dato que se utiliza para almacenar una colección ordenada y mutable de elementos. Se definen utilizando corchetes ([]), y los elementos de la lista se separan por comas. Por ejemplo:

```
lista1 = [1, 2, 3]
lista2 = ['a', 'b', 'c']
lista3 = [0] * 20
```

En estos ejemplos, tanto `lista1` como `lista2` son variables que contienen listas. Se deben utilizar corchetes y separar los elementos por comas para definir una lista. Incluso se puede declarar una lista vacía con los corchetes y sin indicar ningún valor entre ellos. La listad denominada `lista3` esta generada con el operador de producto, el mismo genera una lista de 20 elementos, todos con valor 0.

Las listas son similares a las tuplas, pero tienen la ventaja de ser mutables, lo que significa que se pueden modificar una vez creadas. Esto implica que se puede agregar, eliminar o cambiar elementos individualmente en una lista. Asimismo se puedes acceder a los elementos individuales de una lista utilizando el operador de acceso indexado.

Además de acceder a elementos individuales, las listas en Python admiten una variedad de operaciones y métodos que te permiten modificar, agregar, eliminar y manipular los elementos de la lista. Algunas operaciones comunes incluyen:

```
lista = [1, 2, 3]

#Agregar elementos a una lista con el método append():
lista.append(4)
print(lista)  # Resultado: [1, 2, 3, 4]

#Eliminar elementos de una lista con el método remove():
lista.remove(2)
print(lista)  # Resultado: [1, 2, 4]

# Obtener la longitud de una lista con la función len():
longitud = len(lista)
print(longitud)  # Resultado: 3

# Revertir una lista con el método reverse():
lista.reverse()
print(lista)  # Resultado: [4, 2, 1]

# Ordenar una lista con el método sort():
lista.sort()
print(lista)  # Resultado: [1, 2, 4]
```

2.5. Generación de listas por comprensión

La generación por comprensión, también conocida como comprensión de listas (list comprehension en inglés), es una construcción sintáctica que permite crear listas de manera concisa y eficiente basándose en una expresión y un conjunto de iteraciones o condiciones.

La sintaxis básica de una comprensión de lista es la siguiente:

```
nueva_lista = [expresión for elemento in secuencia]
```

Donde `expresión` representa la expresión o cálculo que se realizará en cada elemento de la secuencia, `elemento` es la variable de iteración que toma el valor de cada elemento en la secuencia, y `secuencia` es la fuente de valores sobre la cual se iterará.

Por ejemplo, dada una lista de números la necesidad de crear una nueva lista que contenga el cuadrado de cada número. Se puede usar una comprensión de lista de la siguiente manera:

```
numeros = [1, 2, 3, 4, 5]
cuadrados = [numero ** 2 for numero in numeros]
print(cuadrados)
# [1, 4, 9, 16, 25]
```

En este ejemplo, la comprensión de lista `[numero ** 2 for numero in numeros]` itera sobre cada elemento `numero` en la lista `numeros` y calcula el cuadrado de cada número utilizando la expresión `numero ** 2`. Los resultados se agregan automáticamente a una nueva lista llamada `cuadrados`.

Además de las iteraciones simples, también es posible incluir condiciones en una comprensión de lista. Por ejemplo, si se desea obtener solo los números pares de una lista, se puede agregar una condición utilizando la cláusula if:

```
numeros = [1, 2, 3, 4, 5]
pares = [numero for numero in numeros if numero % 2 == 0]
print(pares)
# [2, 4]
```

En este caso, la comprensión de lista filtra los números de la lista pero solo incluye aquellos que cumplen la condición, es decir, los números pares.

2.6. Ejercicios

2.6.1. Simulador de ruleta

Se necesita desarrollar un programa que simule el juego de la ruleta.
Para ello se deben generar al azar 1000 tiradas y luego informar:

- Cantidad de pares e impares
- Cantidad de tiradas por cada docena
- Porcentaje de ceros sobre el total de jugadas.
- Cantidad de rojos y de negros

Solución La simulación está programada en la función `simulacion`. Dado que los resultados son numerosos, la función no los retorna sino que los imprime directamente al terminar los cálculos.

Para el conteo de tiradas clasificado por color, el problema principal es el de conocer el color de cada número en una ruleta real. El algoritmo de determinación del color es el siguiente:

- Los números 10 y 28 son negros.
- Los otros números son negros si la suma de sus dígitos es par.
- La suma de los digitos no es la suma simple, si al sumar los dígitos del número el resultado es mayor a 9, deben volver a sumarse los digitos de dicho resultado hasta reducir a un único dígito.

Para ello el programa inicia generando una lista con los colores de los números, almacenando en la misma una tupla en la que el primer elemento es un número y el segundo el color asociado al mismo. En dicha lista las tuplas se almacenan de forma que cada número está almacenado en la lista en su mismo índice, por ejemplo, el número 10 esta almacenado como (10,"N") en la posición 10. La lista es creada en la función `generar_ruleta`.

Para el cálculo del color la función `get_color` recibe un número y devuelve el color correspondiente al parámetro.

Finalmente la función `reducir_numero` recibe un número entero y calcula la suma de sus dígitos. La función se llama recursivamente cuando la suma obtenida es mayor a 9 y por lo tanto es un número de más de un dígito.

ruleta.py

```
"""
Simulador de Ruleta

Desarrollar un programa que simule el juego de la ruleta.

Para ello generar al azar 1000 tiradas y luego informar:

* Cantidad de pares e impares
* Cantidad de tiradas por cada docena
* Porcentaje de ceros sobre el total de jugadas.
* Cantidad de rojos y de negros
"""

import random
from rich import print
from rich.console import Console
from rich.panel import Panel
from rich.table import Table

"""
Reglas:

Comienza en 1,R (1-ROJO)
Los números negros N son aquellos cuya reducción es PAR, o sea, la suma de sus dígitos es divisible por 2
"""

console = Console()

def reducir_numero(numero: int) -> int:
    """La función toma un número entero y devuelve un entero que es igual a la suma de todos sus dígitos.
    Si la suma tiene dos dígitos repite el proceso hasta que tenga un solo dígito.

    :param numero: Número que se desea reducir
    :type numero: int
    :return: Valor de UN dígito, obtenido a partir de la suma de los dígitos del número original.
    :rtype: int
```

```
    """

    # Caso base: Si el número tiene un solo dígito, devolverlo tal cual
    if numero < 10:
        return numero
    # Convertir el número en una lista de dígitos
    digitos = [int(digito) for digito in str(numero)]
    # Calcular la suma de los dígitos
    suma_digitos = sum(digitos)
    # Llamar recursivamente a la función con la suma de los dígitos
    return reducir_numero(suma_digitos)

def get_color (nro: int) -> str:
    """Dado un número de la ruleta, determinar el color de dicho número en el paño

    :param nro: Número del cual se desea averiguar el color
    :type nro: int
    :return: Color del número indicado. Valores válidos: V (verde), R (rojo), N (negro)
    :rtype: str
    """
    if nro == 0:
        return 'V'
    elif nro == 10 or nro == 28:
        return 'N'
    else:
        # Los numeros negros son los pares
        # Aquellos cuya reducción es par
        # El 10 y el 28
        if reducir_numero(nro)%2 == 0:
            return 'N'
        else:
            return 'R'

def generar_ruleta():
    """Genera un tablero de ruleta, como una lista de tuplas conteniendo (numero,color)
    """
    ruleta = []
    for nro in range (0,37):
        ruleta.append((nro,get_color(nro)))
    return ruleta

def imprimir_ruleta(rul: list[(int,str)]):
    """Imprime por consola un paño de ruleta, obtenido por medio de una lista de tuplas

    :param rul: lista que representa el paño de la ruleta
    :type rul: lista de tuplas (numero, color)
    """
    col = 1
    for nro,color in rul:
        if color == 'V':
            console.print (f"[bold white on green]{nro:^12}[/]", end="")
            print()
        elif color == 'R':
            console.print (f"[bold white on red]{nro:^4}[/]", end="")
            col += 1
        else:
            console.print (f"[bold white on black]{nro:^4}[/]", end="")
            col += 1

        if col == 4:
            print()
            col = 1

def imprimir_tirada (nro : int, color : str):
    if color == 'V':
        console.print (f"Obtuvimos [bold white on green]{nro}{color}[/]")
    elif color == 'R':
        console.print (f"Obtuvimos [bold white on red]{nro}{color}[/]")
    else:
        console.print (f"Obtuvimos [bold white on black]{nro}{color}[/]")

def simulacion(ruleta: list[(int,str)]):
    """Realiza una simulación y obtiene indicadores sobre la misma

    :param ruleta: Ruleta generada, para determinar los colores de los numeros
    :type ruleta: lista de tuplas (numero,color)
    """
```

```
    cantidad_pares = cantidad_impares = 0
    cantidad_rojos = cantidad_negros = 0
    tiradas_primera_docena = tiradas_segunda_docena = tiradas_tercera_docena = 0
    cantidad_ceros = 0
    jugadas = 1000

    # Generar 1000 numeros enteros aleatorios en el rango de la ruleta (0-36)
    for i in range(jugadas):
        tirada_nro = random.randint(0,36)
        tirada_color = ruleta[tirada_nro][1]
        #imprimir_tirada(tirada_nro,tirada_color)

        if tirada_nro%2 == 0:
            cantidad_pares += 1
        else:
            cantidad_impares += 1

        if tirada_color == 'R':
            cantidad_rojos += 1
        elif tirada_color == 'N':
            cantidad_negros += 1
        else:
            cantidad_ceros += 1

        docena = tirada_nro // 12
        if docena == 1:
            tiradas_primera_docena += 1
        elif docena == 2:
            tiradas_segunda_docena += 1
        else:
            tiradas_tercera_docena += 1
    porcentaje_ceros = cantidad_ceros*100/jugadas

    console.print()
    table = Table(title="Resultados de la simulación")
    table.add_column("Resultado")
    table.add_column("Valor",justify="center")
    table.add_row("Cantidad de pares",str(cantidad_pares))
    table.add_row("Cantidad de impares",str(cantidad_impares))
    table.add_row("Cantidad de tiradas 1° docena",str(tiradas_primera_docena))
    table.add_row("Cantidad de tiradas 2° docena",str(tiradas_segunda_docena))
    table.add_row("Cantidad de tiradas 3° docena",str(tiradas_tercera_docena))
    table.add_row("Porcentaje de ceros",str(porcentaje_ceros)+"%")
    table.add_row("Cantidad de rojos","[bold white on red]"+str(cantidad_rojos))
    table.add_row("Cantidad de negros","[bold white on black]"+str(cantidad_negros))
    console.print(table)

def principal():
    ruleta = generar_ruleta()
    imprimir_ruleta(ruleta)
    simulacion(ruleta)

if __name__ == "__main__":
    principal()
```

Capítulo 3

Archivos

En muchos casos, los programas requieren almacenar datos de forma persistente, de manera que estos datos se mantengan entre distintas ejecuciones del programa.

Los archivos son el mecanismo más básico y ampliamente utilizado en cualquier lenguaje de programación para lograr esta persistencia de datos.

Los archivos difieren significativamente de las bases de datos en el sentido de que son universales, lo que significa que cualquier programa que comprenda el formato de los datos almacenados en un archivo puede leerlos e incluso escribir sobre ellos, sin importar qué programa los haya generado previamente.

Existen notables diferencias entre los archivos y las bases de datos. Las bases de datos ofrecen ventajas significativas en términos de velocidad y capacidad de manipulación de grandes cantidades de datos. Además, proporcionan características de seguridad, como mecanismos de autenticación y autorización, para gestionar datos sensibles. Otro aspecto destacado de las bases de datos es su capacidad para mantener la consistencia de los datos, evitando la presencia de información incoherente o inconsistente.

A pesar de estas ventajas, la manipulación de datos a través de archivos ha sido ampliamente utilizada históricamente para transmitir información entre sistemas heterogéneos. Esta práctica es común cuando se requiere transferir datos de un programa a otro que no puede comunicarse directamente entre sí. Los archivos permiten que un programa almacene datos en un archivo y que otro programa, incluso escrito en un lenguaje o sistema diferente, pueda manipular esos datos. Este mecanismo ha sido tradicionalmente utilizado y sigue siendo utilizado en el campo de la ciencia de datos, donde se requiere la transferencia y manipulación conjunta de datos provenientes de diferentes fuentes.

3.1. Tipos de archivos

Cuando se requieren manipular datos almacenados en archivos, es fundamental considerar el formato o tipo de archivo. Existen dos tipos principales de archivos: archivos de texto y archivos binarios.

Los archivos de texto contienen información representada como una secuencia de caracteres legibles, escritos en una determinada codificación (UTF-8, ASCII, etc.). Estos archivos utilizan caracteres especiales, como el retorno de carro o la nueva línea, para delimitar cada línea y permitir la manipulación individual de cada dato. El formato de archivo de texto es adecuado para almacenar datos legibles por humanos, como texto plano, configuraciones o registros de datos en formato tabular.

Por otro lado, los archivos binarios no dependen de los caracteres de retorno de carro para delimitar los datos. En cambio, requieren de algún mecanismo específico para indicar la forma en que los datos están organizados dentro del archivo.Los archivos binarios son archivos en los que la información se almacena en un formato no legible directamente por los humanos y pueden contener datos de diversos tipos, como números, imágenes, audio, video, entre otros, donde la representación en formato de texto no sería práctica o eficiente. A diferencia de los archivos de texto, los archivos binarios no están codificados como caracteres legibles, sino que utilizan una representación binaria de los datos.

En resumen, los archivos de texto son adecuados para almacenar datos en formato de texto legible, mientras que los archivos binarios son utilizados para guardar datos más complejos y diversos, sin requerir la conversión a cadenas de texto.

3.2. Apertura y cierre

Para interactuar con un archivo en Python, es necesario informar al sistema operativo qué archivo se pretende utilizar, dónde está ubicado y qué operaciones se van realizar en él. Para lograr esto, se debe realizar la operación de apertura del archivo, indicando al sistema operativo que el programa está utilizando ese archivo y, potencialmente, evitando que otros programas accedan a él simultáneamente. Al finalizar el uso del archivo, es importante cerrarlo.

El cierre de un archivo cumple dos propósitos básicos. En primer lugar, notifica al sistema operativo que ya no se está utilizando y, por lo tanto, otros programas pueden acceder a él. En segundo lugar, en el caso de operaciones de escritura de datos en el archivo, el cierre garantiza que las últimas escrituras se almacenen correctamente en el disco o en el medio de almacenamiento utilizado. Al realizar escrituras, es probable que la biblioteca de manipulación de archivos o el propio sistema operativo espere a acumular varias operaciones de escritura antes de almacenarlas en el dispositivo de almacenamiento. Si un archivo no se cierra correctamente, es posible que las últimas escrituras no se envíen al destino, por lo que es importante cerrar siempre los archivos.

Python nos proporciona dos funciones para estas operaciones, llamadas precisamente `open` y `close`. La función open recibe dos parámetros. El primero es el nombre del archivo, una cadena que contiene el nombre y la extensión del archivo, y opcionalmente la ubicación utilizando rutas relativas o absolutas para especificar el directorio donde se encuentra el archivo. El segundo parámetro es el modo de apertura, donde se indica el tipo de archivo (texto o binario) y la operación que deseamos realizar en dicho archivo.

Open devuelve una estructura de datos que nos permite operar con el archivo. Esta estructura cuenta con un método llamado close, que se utiliza para indicar que el archivo ha sido cerrado correctamente.

En general, es recomendable mantener los archivos abiertos el menor tiempo posible. Por lo tanto, se debe abrir el archivo, realizar las operaciones necesarias y cerrarlo inmediatamente. No debe dejarse un archivo abierto más tiempo del necesario, especialmente durante el ingreso de datos por parte del usuario.

3.3. Modos de apertura

El modo de apertura que se especifica como parámetro en la función open es una cadena de texto que puede contener una o dos letras. Estas letras indican la operación que se desea realizar en el archivo. Para la operación de lectura, se utiliza la letra “R” (read), mientras que para la operación de escritura se utiliza la letra “W” (write).

El modo de apertura “W” tiene la particularidad de que, si el archivo en el que se desea escribir no existe, se crea automáticamente. Sin embargo, si el archivo ya existe, su contenido se borra por completo y se empieza a escribir desde cero, es decir, como un archivo vacío.

Existe otro modo de apertura para escritura denominado “X”. Este modo permite escribir datos en un archivo, pero si el archivo ya existe, impide borrar su contenido original y arroja un error. En resumen, el modo “X” es utilizado para escribir en un archivo completamente nuevo sin sobrescribir uno existente.

Otro modo de escritura es el modo “A” (append), que significa agregar. Cuando se abre un archivo en modo “A”, si el archivo no existe, se crea. Sin embargo, si el archivo ya existe, el contenido previo no se borra, sino que se comienza a escribir a continuación del último dato almacenado en el archivo.

Por último, existe el modo “R+” que permite tanto la lectura como la escritura en un archivo. Al abrir un archivo en modo “R+”, todas las operaciones de lectura y escritura se realizan desde el principio del archivo, aunque con operaciones de desplazamiento se puede indicar una posición específica para continuar a partir de ella.

Es importante mencionar que también se debe indicar el tipo de archivo con una letra “T” para archivos de texto y una letra “B” para archivos binarios. Si no se especifican estas letras, el modo de apertura predeterminado es de lectura de archivos de texto (“RT”).

Modo	Descripción	Acción sobre el archivo
`r`	Lectura	Abre el archivo para lectura (modo texto)
`w`	Escritura	Abre el archivo para escritura (modo texto)
`x`	Escritura exclusiva	Abre el archivo para escritura exclusiva (modo texto)
`a`	Adjuntar	Abre el archivo para agregar contenido al final (modo texto)
`r+`	Lectura y escritura	Abre el archivo para lectura y escritura (modo texto)
`rb`	Lectura binaria	Abre el archivo para lectura (modo binario)
`wb`	Escritura binaria	Abre el archivo para escritura (modo binario)
`xb`	Escritura exclusiva binaria	Abre el archivo para escritura exclusiva (modo binario)
`ab`	Adjuntar binario	Abre el archivo para agregar contenido al final (modo binario)
`r+b`	Lectura y escritura binaria	Abre el archivo para lectura y escritura (modo binario)

3.4. Lectura de archivos de texto

Para leer archivos de texto, hay tres métodos disponibles en la estructura devuelta por la función open. El primer método es `read`, que lee todo el contenido del archivo y devuelve una variable de tipo string. Este método es adecuado para archivos de texto relativamente pequeños, ya que carga todo el contenido en una sola variable. La principal desventaja es que, al leer todo el contenido, es responsabilidad del programador manipular el formato si el archivo tiene una estructura interna. Por ejemplo, si el archivo está separado por comas, será responsabilidad del programa dividir los datos según los delimitadores. Si el archivo contiene varias líneas de texto, read las lee todas y devuelve una cadena que incluye los caracteres de nueva línea o retorno de carro que indican el final de cada línea.

El segundo método es `readline`, que lee una sola línea y deja el archivo en un estado en el que la siguiente llamada leerá la línea que se encuentre a continación. De esta manera, readline devuelve una cadena con el contenido de una línea y permite la lectura en un ciclo para procesar líneas sucesivas. Este es el método más adecuado cuando se trata de archivos relativamente grandes, ya que al leer las líneas una a una, no se carga todo el contenido del archivo en la memoria, sino que se mantiene una única línea en memoria a la vez.

Por último, el método `readlines` (en plural) recorre todo el archivo y lee cada una de las líneas, devolviendo una lista donde cada posición contiene una línea de texto. Si se lo utiliza en un archivo con, por ejemplo, 20 líneas, el mismo devolverá una lista con 20 elementos, cada uno representando una línea del archivo en el orden en que fueron leídas. Readlines se encarga de dividir el archivo según los caracteres de nueva línea, pero ocupa más memoria, ya que todo el contenido del archivo se almacena en la lista. Por lo tanto, no sería apropiado para leer archivos de texto demasiado grandes, en el orden de varios megabytes, ya que consumiría una cantidad considerable de memoria.

Por lo tanto, para leer todo el contenido de un archivo de texto en una variable e imprimirla, se puede programar el siguiente bloque:

```
archivo = open("datos.txt")
contenido = archivo.read()
print(contenido)
archivo.close()
```

Y para leer un archivo más largo linea a linea:

```
archivo = open("datos.txt")

linea = archivo.readline()
while linea:
    print(linea)
    linea = archivo.readline()

archivo.close()
```

En este ejemplo, se abre el archivo “datos.txt” en modo lectura. Luego se utiliza un ciclo while para leer cada línea del archivo utilizando el método readline(). Dentro del ciclo, se imprime la línea y se lee la siguiente línea. El ciclo continúa hasta que se alcanza el final del archivo, es decir, cuando el método readline() devuelve una cadena vacía. Por último, se cierra el archivo utilizando el método close().

3.5. Escritura de archivos de texto

Para la escritura de archivos de texto existen dos métodos principales. El primer método es `write`, que permite escribir una cadena en el archivo. Al utilizar este método, la cadena se escribe en el archivo sin incluir automáticamente caracteres de nueva línea. Si se necesita escribir varias líneas de texto, es responsabilidad del programador agregar los caracteres de nueva línea en los lugares adecuados dentro de la cadena.

El segundo método es `writelines`, que recibe una lista de cadenas y escribe cada una de ellas en el archivo de texto en el orden en que aparecen en la lista. Al igual que el método write,writelines no agrega automáticamente caracteres de nueva línea, por lo que es responsabilidad del programador asegurarse de que las cadenas de la lista incluyan los caracteres de nueva línea donde sea necesario.

Por ejemplo:

```
nombres = ['Juan', 'María', 'Carlos', 'Laura']

archivo = open("nombres.txt", "w")

for nombre in nombres:
    archivo.write(nombre + '\n')

archivo.close()
```

En este ejemplo, se crea una lista llamada `nombres` con algunos nombres. Luego se abre el archivo “nombres.txt” en modo escritura. A continuación, se utiliza un bucle for para recorrer cada elemento de la lista nombres. En cada iteración, se utiliza el método write() para escribir el nombre en el archivo, seguido de un carácter de nueva línea \\`n`. Esto garantiza que cada nombre se escriba en una línea nueva del archivo.

3.6. Ejercicios

3.6.1. Lectura de un archivo de números

Desarrollar un programa que lea todo el contenido del archivo numeros.txt que contiene múltiples líneas de texto, cada una de ellas con un número entero. Al leer el archivo se debe almacenar todos los números en una lista. A continuación el programa debe manipular los números cargados en la lista para:

- Calcular e imprimir el promedio de todos los números
- Calcular e imprimir la cantidad de números mayores al promedio
- Generar y mostrar una nueva lista que contenga todos los números pares

proceso_listas.py

```
def cantidad_menor(lista, techo):
    """
    Cuenta los elementos de una lista que sean menores a un tope
```

```
    :param lista: lista de valores
    :type lista: lista de datos que soporte comparación por menor
    :param techo: valor máximo para filtrar
    :type techo: el mismo de los elementos de la lista
    :returns: la cantidad de elementos cuyo valor sea menor al techo
    :rtype: entero
    """
    c = 0
    for x in lista:
        if x < techo:
            c += 1

    return c

def cantidad_mayor(lista, piso):
    """
    Cuenta los elementos de una lista que sean menores a un tope

    :param lista: lista de valores
    :type lista: lista de datos que soporte comparación por menor
    :param techo: valor máximo para filtrar
    :type techo: el mismo de los elementos de la lista
    :returns: la cantidad de elementos cuyo valor sea menor al techo
    :rtype: entero
    """
    c = 0
    for x in lista:
        if x > piso:
            c += 1

    return c


def promedio(lista):
    """
    Calcula el promedio simple de todos los elemento de una lista

    :param lista: lista
    :type lista: lista de números
    :returns: el promedio de todos los valores de la lista
              o 0 si la lista está vacía
    :rtype: flotante
    """
    cantidad = len(lista)
    if cantidad == 0:
        return 0

    suma = 0
    for x in lista:
        suma += x

    return suma / cantidad


def existe(lista, buscado):
    """
    Busca un valor en una lista e informa si lo pudo encontrar

    :param lista: una lista de cualquier tipo
    :param buscado: el valor que se busca
    :tipo buscado: el mismo tipo que el de los datos de la lista
    :returns: verdadero si el elemento buscado existe en la lista
              y falso en caso contrario
    :rtype: boolean
    """
    for x in lista:
        if x == buscado:
            return True

    return False
```

lectura_numeros.py

```
from proceso_listas import *

def leer_archivo(nombre_archivo):

    numeros = []

    archivo = open(nombre_archivo)

    while True:
        linea = archivo.readline()
        if not linea:
            break
        n = int(linea[:-1])
        numeros.append(n)

    archivo.close()

    return numeros


if __name__ == "__main__":
    lista = leer_archivo("numeros.txt")
    promedio_todos = promedio(lista)
    mayores = cantidad_mayor(lista, promedio_todos)
    pares = [x for x in lista if x % 2 == 0]

    print("Del archivo se leyeron los números")
    print(f"El promedio de esos números es de {promedio_todos:5.2f}")
    print(f"Y {mayores} son mayores que el promedio")
    print("Listado de los pares:")
    for x in pares:
        print(x)
```

3.6.2. Lectura de un csv con códigos postales

El archivo cp.csv contiene el listado de los códigos postales de tres provincias argentinas. En cada línea se encuentran, separados por un símbolo de punto y coma (;) los siguientes datos:

- Provincia: representada por una letra mayúscula según el estándar ISO correspondiente.
- Código: es un número entero de cuatro dígitos que identifica una localidad o varias localidades vecinas. Puede estar repetido.
- Nombre: es el nombre de la localidad correspondiente a ese código.

Se requiere leer todo el contenido del archivo y guardarlo en una lista que contenga un elemento por cada línea. En cada elemento debe almacenarse alguna estructura de datos que permita acceder individualmente a cada dato que conforma un codigo postal.

Luego de la carga el programa debe permitir el ingreso de uno o más códigos numéricos y listar la provincia y nombre de todas las localidades asignadas a dichos códigos

cp.csv (primeras líneas)

```
K;4743;ACONQUIJA
K;4701;ACOSTILLA
K;4234;ACHALCO
K;5263;ADOLFO E. CARRANZA
K;4139;AGUA AMARILLA (LA HOYADA, DPTO. SANTA MARIA)
```

lectura_cp.py

```
def leer_codigos(nombre_archivo):

    codigos = []
    archivo = open(nombre_archivo)

    for linea in archivo.readlines():
        codigo = tuple(linea[:-1].split(";"))
        codigos.append(codigo)

    archivo.close()

    return codigos


def buscar_codigo(lista, buscado):
    encontrados = []

    for c in lista:
        if c[1] == buscado:
            encontrados.append(c)

    return encontrados
    # Con comprensión de listas
    # return [c for c in lista if c[1] == buscado]

if __name__ == "__main__":
    codigos = leer_codigos("cp.csv")

    buscado = input("Ingrese un código a buscar (fin con cadena vacía):")
    while buscado:
        encontrados = buscar_codigo(codigos, buscado)
        if encontrados:
            for c in encontrados:
                print(f"{c[0]}: {c[2]}")
        else:
            print("No se encontró ninguna localidad")
        buscado = input("Ingrese un código a buscar (fin con cadena vacía):")
```

3.6.3. Análisis del archivo de propinas

El archivo tips.csv contiene los datos de las propinas recibidas por los empleados de una pizzería en New York. El archivo contiene los siguientes campos separados por comas

- total_bill: Importe total del servicio
- tip: Importe de la propina
- sex: Sexo del cliente
- smoker: El cliente es fumador (Yes, No)
- day: Día (Juev, Vie, Sab, Dom)
- time: Horario (Almuerzo, Cena)
- size: Tamaño de la mesa (comensales)

Se requiere desarrollar un programa que desde los datos del archivo informe:

- ¿Quién paga más propinas?¿Hombres o mujeres?
- ¿Cuáles son los días más 'lentos', con menos propinas?
- ¿Cuál es el promedio de las propinas?
- ¿Cuándo se vendió la orden más grande (qué día y en qué turno)?

tips.csv (primeras líneas)

```
total_bill,tip,sex,smoker,day,time,size
16.99,1.01,Mujer,No,Dom,Cena,2
10.34,1.66,Hombre,No,Dom,Cena,3
21.01,3.5,Hombre,No,Dom,Cena,3
23.68,3.31,Hombre,No,Dom,Cena,2
```

tips.py

```
propinas_hombres = propinas_mujeres = 0
propina_fumador = propina_no_fumador = 0
propinas_acum = [0]*4
promedio = 0
cant_operaciones = 0
dia_orden_mayor = None
turno_orden_mayor = None
monto_orden_mayor = None

########################################################################
# Para utilizar la librería mejorada de consola (rich) primero hay que
# instalarla utilizando el comando pip install rich
########################################################################
from rich import print
from rich.console import Console
from rich.panel import Panel

c = Console()
c.clear()

first_row = True
with open("tips.csv","r") as file:
    for fila in file:
        # Debo saltear el primer registro ya que contiene los titulos
        if first_row:
            # Indicar que ya no estoy en la primera fila
            first_row = False
        else:
            cant_operaciones += 1
            # Hacer todo el procesamiento
            # Ahora debo separar la fila leída en sus componentes
            campos = fila[:-1].split(",")
            # print(campos)
            # Saco los campos que necesito
            total = float(campos[0])
            propina = float(campos[1])
            sexo = campos[2]
            dia = campos[4]
            horario = campos[5]

            # Acumular por separado las propinas de hombres y mujeres
            if sexo == 'Hombre':
                propinas_hombres += propina
            else:
                propinas_mujeres += propina

            # Obtener los datos de la mayor orden
            # Cuándo? Cuando sea la primera o sea mayor a la anterior
            if monto_orden_mayor == None or total > monto_orden_mayor:
                monto_orden_mayor = total
                dia_orden_mayor = dia
                turno_orden_mayor = horario

            # Solo trabajo 4 días así que puedo acumular las propinas
            # en un vector de 4 elementos, relacionados
            # a Jue, Vie, Sab, Dom
            if dia == 'Juev':
                propinas_acum[0] += propina
            elif dia == 'Vie':
                propinas_acum[1] += propina
            elif dia == 'Sab':
                propinas_acum[2] += propina
            else:
                propinas_acum[3] += propina

# Calcular la propina promedio
promedio = (propinas_hombres+propinas_mujeres)/cant_operaciones

# Recorrer el vector de propinas y ver cual es el menor
menor_indice = 0  # Voy a suponer que es el primero
menor_propina = propinas_acum[0]
menor_dia = ""

for i in range(len(propinas_acum)):
    if propinas_acum[i] < menor_propina:
```

```
        menor_propina = propinas_acum[i]
        menor_indice = i
# Cuando termine tengo que ver a qué dia corresponde ese índice
if menor_indice == 0:
    menor_dia = "Jueves"
elif menor_indice == 1:
    menor_dia = "Viernes"
elif menor_indice == 2:
    menor_dia = "Sábado"
else:
    menor_dia = "Domingo"

c.print(Panel("Análisis de propinas en New York"))

print (f"[white bold on green]Propinas de los hombres:[/] {propinas_hombres:.2f}")
print (f"[white bold on green]Propinas de las mujeres:[/] {propinas_mujeres:.2f}")
print (f"[white bold on green]Propina promedio:[/] {promedio:.2f}")
print (f"La mayor venta fué de [on green]{monto_orden_mayor}[/] y se realizó un [on green]{dia_orden_mayor}[/] en
↪  el turno [on green]{turno_orden_mayor}[/]")
print (f"Los días más lentos son los [on green]{menor_dia}[/] con solamente [on green]{menor_propina}[/] de
↪  propina")
c.print(Panel("Fin de la ejecución"))
```

Capítulo 4

Colecciones avanzadas

4.1. Conjuntos

4.1.1. Introducción

Los conjuntos son una estructura de datos proporcionada por el lenguaje que permite almacenar y manipular colecciones de elementos. A diferencia de las listas o las tuplas, los conjuntos tienen características distintivas que los hacen únicos.

Una de las principales características de los conjuntos es su capacidad para eliminar automáticamente elementos duplicados. Esto significa que, al insertar un elemento repetido en un conjunto, este será ignorado y no se agregará nuevamente.

Otra diferencia importante es que los conjuntos no admiten acceso indexado. No es posible acceder a elementos individuales utilizando el operador de corchetes o rebanadas, como se puede hacer en las listas o tuplas.

Sin embargo, la característica principal de los conjuntos es su eficiencia en las operaciones de búsqueda. Mientras que en las listas o tuplas las búsquedas pueden ser lentas, ya que requieren recorrer todos los elementos, en los conjuntos se utiliza una estrategia de dispersión que agiliza significativamente esta tarea. La estructura de datos desordena los elementos de manera inteligente, permitiendo localizar rápidamente un elemento en base a su criterio de dispersión, sin necesidad de recorrer todos los elementos.

Esta propiedad hace que los conjuntos sean especialmente útiles para realizar búsquedas de elementos sin duplicados. Además, los conjuntos son ideales para realizar operaciones de conjuntos matemáticos, como uniones, intersecciones y diferencias. Estas operaciones, que requerirían una programación relativamente compleja y lenta con listas o tuplas, pueden ser realizadas de manera eficiente y sencilla con conjuntos.

En resumen, los conjuntos son una valiosa estructura de datos que permiten almacenar y manipular colecciones de elementos sin duplicados. Su eficiencia en las operaciones de búsqueda y su capacidad para realizar operaciones de conjuntos de manera rápida los convierten en una herramienta poderosa para resolver diversos problemas.

4.1.2. Operaciones de datos

Creación

Existen tres formas de crear conjuntos en Python. La primera opción es utilizar la función `set()`, que crea un conjunto vacío. Este conjunto vacío se puede asignar a una variable para su posterior manipulación. Una vez creado el conjunto vacío, se pueden utilizar métodos para insertar elementos en él.

La segunda opción es crear un conjunto mediante la enumeración de elementos conocidos. Esto se asemeja a la notación matemática, donde se definen los elementos del conjunto entre llaves y separados por comas. Por ejemplo: `{1, 2, 3}`.

La tercera opción es utilizar comprensiones de conjuntos, que son similares a las comprensiones de listas. Se utiliza la misma sintaxis de comprensión, pero se delimita con un par de llaves en lugar de un par de corchetes. En ambos casos, es importante tener en cuenta que los conjuntos no permiten elementos duplicados, por lo que la cantidad de elementos en el conjunto resultante puede ser menor a la cantidad de elementos especificados en la creación. Los elementos duplicados serán ignorados. Por ejemplo: `{random.randint(1,10000) for x in range(100)}` crea un conjunto con hasta 100 números al azar sin repeticiones.

Inserción

Una vez que se ha creado un conjunto utilizando una de las tres alternativas mencionadas anteriormente, es posible agregar nuevos valores al conjunto utilizando dos métodos específicos.

El primer método es `add()`, que permite agregar un nuevo elemento al conjunto verificando que no exista previamente. En caso de que el elemento ya esté presente en el conjunto, el método add() no generará ninguna acción ni devolverá ningún error. El conjunto permanecerá sin modificaciones. Sin embargo, si el elemento que se intenta agregar no se encuentra en el conjunto, el método add() lo añadirá correctamente, expandiendo el conjunto con un elemento adicional.

El segundo método es `update()`, el cual recibe como parámetro una secuencia iterable, como un rango, una cadena, una tupla, una lista o incluso otro conjunto. Este método se encarga de recorrer todos los elementos de la secuencia iterable y los inserta en el conjunto original. Sin embargo, al igual que el método add(), el método update() garantiza que solo se insertarán elementos que no estén duplicados y que no existan previamente en el conjunto. Esto significa que no se agregarán elementos repetidos al conjunto durante el proceso de actualización.

Eliminación

Además de agregar elementos a un conjunto, también es posible eliminar elementos utilizando varios métodos disponibles.

El primer método es `remove()`, el cual recibe como parámetro el valor que se desea eliminar del conjunto. Este método elimina el elemento especificado y, en caso de que el elemento no exista en el conjunto, generará un error. Es importante tener cuidado al utilizar remove() para asegurarse de que el elemento a eliminar realmente esté presente en el conjunto.

El segundo método es `discard()`, que también recibe como parámetro el valor que se desea eliminar. Si el elemento existe en el conjunto, discard() lo eliminará sin generar errores. Sin embargo, si el elemento no está presente, discard() simplemente ignorará la solicitud y permitirá que el programa continúe sin interrupciones.

Otra opción para eliminar elementos es el método `pop()`, el cual selecciona y elimina arbitrariamente un elemento del conjunto. El elemento eliminado se devuelve como resultado de la llamada a pop(), lo que permite al usuario conocer qué elemento fue eliminado en caso de necesitarlo.

Por último, el método `clear()` borra todos los elementos del conjunto, dejándolo vacío. Este método es útil cuando se desea reiniciar el conjunto y eliminar todos sus elementos sin eliminar la variable en sí.

Recorridos

Cuando se trabaja con conjuntos y se desea recorrer sus elementos, se puede utilizar la instrucción for de manera similar a como se recorren las listas. Con la sintaxis for variable in conjunto, se realizará una iteración por cada elemento almacenado en el conjunto, asignando cada valor a la variable del ciclo for.

Es importante tener en cuenta que al recorrer un conjunto, los elementos no se encontrarán en un orden específico. A diferencia de las listas, donde los elementos se mantienen en el orden en que se insertaron, el recorrido de un conjunto mostrará los elementos completamente mezclados y desordenados. Esto se debe a que los conjuntos utilizan una estrategia de dispersión para lograr búsquedas rápidas, lo cual implica que los datos se almacenan de manera dispersa y desordenada.

Si se requiere recorrer un conjunto en un orden específico, como un orden numérico o alfabético, no es posible lograrlo directamente con un conjunto. Sin embargo, los conjuntos disponen del método `sorted()`, el cual devuelve una nueva lista con los mismos elementos del conjunto, pero ordenados según sus valores. Es importante destacar que esto implica la creación de una nueva estructura de datos (la lista ordenada) y no afecta el orden interno del conjunto.

A veces, al recorrer un conjunto, puede parecer que los elementos están ordenados, especialmente si se insertan solo números enteros. Sin embargo, esto no se puede garantizar para otros tipos de datos o para conjuntos más grandes. Por lo tanto, nunca se debe confiar en que un conjunto tendrá los datos ordenados, a menos que se utilice el método sorted() para obtener una lista ordenada explícitamente.

4.1.3. Operaciones de conjuntos

Los conjuntos son especialmente útiles para eliminar elementos duplicados, ya que garantizan la unicidad de los datos almacenados. Además, permiten realizar operaciones de conjuntos similares a las que se utilizan en matemáticas.

Pertenencia

La operación principal en un conjunto es verificar la pertenencia de un elemento. Para esto, se utiliza el operador `in`, que verifica si un valor dado pertenece al conjunto. El operador in devuelve True si el elemento está presente en el conjunto y False en caso contrario. La diferencia principal con respecto a las listas es que la velocidad de búsqueda con el operador in no es proporcional a la cantidad de elementos almacenados en el conjunto. Incluso si el conjunto es muy grande, la verificación de pertenencia se realizará de manera eficiente.

Cuando se trabaja con más de un conjunto, se pueden realizar operaciones de conjuntos, como la unión, la intersección y la diferencia. Estas operaciones se aplican utilizando diferentes métodos.

Unión

La unión de conjuntos se puede lograr mediante el método `union()`. Este método recibe como parámetro un segundo conjunto y devuelve un nuevo conjunto que contiene todos los elementos no repetidos de ambos conjuntos. También está disponible el método update(), que realiza la unión modificando el conjunto original.

Intersección

La intersección de conjuntos se puede lograr utilizando el método `intersection()`. Este método recibe como parámetro un segundo conjunto y devuelve un nuevo conjunto con los valores que existen en ambos conjuntos. Además, el método `intersection_update()` realiza la intersección modificando el conjunto original.

Diferencias

La diferencia de conjuntos se puede obtener mediante el método `difference()`. Este método se aplica a un conjunto y recibe como parámetro otro conjunto. Retorna un nuevo conjunto con los elementos presentes en el primer conjunto pero no en el segundo. También está disponible el método `difference_update()`, que realiza la diferencia modificando el conjunto original.

La diferencia simétrica, también conocida como diferencia exclusiva, se obtiene utilizando el método `symmetric_difference()`. Este método recibe como parámetro un segundo conjunto y devuelve un nuevo conjunto que contiene todos los elementos presentes en ambos conjuntos, excepto aquellos que existen en ambos conjuntos al mismo tiempo. El método `symmetric_difference_update()` realiza la diferencia simétrica modificando el conjunto original.

4.1.4. Casos de aplicación

4.1.5. Eliminación de repetidos de una lista

En este ejemplo se dispone de una lista llamada `lista` que contiene elementos duplicados. Para eliminar los duplicados, se convierte la lista en un conjunto utilizando la función `set()`. Los conjuntos en Python sólo pueden contener elementos únicos, por lo que al convertir la lista en un conjunto, automáticamente se eliminan los elementos duplicados.

Luego se convierte nuevamente el conjunto a una lista utilizando la función `list()`y se asigna en una nueva lista llamada `lista_sin_duplicados` que contiene los elementos de la lista original sin duplicados.

Finalmente, se imprime la lista sin duplicados para verificar el resultado.

```
# Lista con elementos duplicados
lista = [1, 2, 3, 4, 2, 3, 5, 6, 1, 7, 8, 5, 9]

# Convertir la lista en un conjunto
# para eliminar duplicados
conjunto = set(lista)

# Convertir el conjunto nuevamente a una lista
lista_sin_duplicados = list(conjunto)

# Imprimir la lista sin duplicados
print(lista_sin_duplicados)

#[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.1.6. Operaciones entre múltiples conjuntos

En este ejemplo se dispone de dos conjuntos `curso_a` y `curso_b` que representan los nombres de personas inscriptas en dos cursos diferentes y sobre ellos se utilizan las siguientes operaciones de conjuntos:

La operación de unión (`union()`) combina ambos conjuntos y devuelve un nuevo conjunto con todos los elementos únicos de ambos conjuntos.

La operación de intersección (`intersection()`) encuentra los elementos comunes entre ambos conjuntos y devuelve un nuevo conjunto con esos elementos.

La operación de diferencia (`difference()`) encuentra los elementos que están en el primer conjunto pero no en el segundo conjunto y devuelve un nuevo conjunto con esos elementos.

La operación de diferencia simétrica (`symmetric_difference()`) encuentra los elementos que están en uno de los conjuntos pero no en ambos conjuntos y devuelve un nuevo conjunto con esos elementos.

Cada operación se aplica a los conjuntos y el resultado se almacena en una variable correspondiente. A continuación se imprimen los conjuntos resultantes para ver los nombres de las personas según cada operación.

```
# Conjunto de nombres de personas del curso A
curso_a = {"Juan", "María", "Pedro", "Luisa", "Ana"}

# Conjunto de nombres de personas del curso B
curso_b = {"Pedro", "Ana", "Sofía", "Carlos"}

# Unión: personas inscriptas en al menos uno de los cursos
union = curso_a.union(curso_b)
print(union)
# {'Juan', 'Carlos', 'María', 'Pedro', 'Luisa', 'Ana', 'Sofía'}

# Intersección: personas inscriptas en ambos cursos
interseccion = curso_a.intersection(curso_b)
print(interseccion)
# {'Ana', 'Pedro'}

# Diferencia: inscriptos sólo en el curso A
diferencia = curso_a.difference(curso_b)
print(, diferencia)
# {'María', 'Luisa', 'Juan'}

# Diferencia simétrica: inscriptos en
# uno de los cursos, pero no en ambos
diferencia_simetrica = curso_a.symmetric_difference(curso_b)
print(diferencia_simetrica)
# {'Carlos', 'Sofía', 'María', 'Juan', 'Luisa'}
```

4.2. Diccionarios

4.2.1. Introducción

Los diccionarios son una estructura de datos altamente versátil y poderosa que permite almacenar información de manera distinta a otras estructuras disponibles en el lenguaje. A diferencia de estas, los diccionarios no almacenan conjuntos o grandes volúmenes de datos, sino que se basan en pares de datos conocidos como “clave” y “valor”. También denominados “arreglos asociativos” en otros lenguajes, los diccionarios establecen una relación entre una clave única en todo el conjunto y un valor asociado a esa clave.

En esencia, los diccionarios almacenan pares de datos, donde la clave actúa como identificador único para un dato específico, y el valor corresponde al dato asociado a esa clave. Este concepto es similar al de las claves primarias utilizadas en estructuras de bases de datos relacionales. Una característica destacada de los diccionarios es la velocidad excepcional de las operaciones de búsqueda. Estas operaciones son tan rápidas como en los conjuntos, lo que implica que la velocidad de búsqueda se mantiene constante, independientemente de la cantidad de datos almacenados. En un diccionario, incluso con una gran cantidad de datos, las búsquedas se realizan a la misma velocidad que si hubiera pocos datos.

Al igual que los conjuntos, los diccionarios no tienen un índice que establezca un orden o posición específica para cada dato y no permiten claves duplicadas. Esta característica es fundamental para su funcionalidad y eficiencia en la recuperación de datos asociados a claves específicas.

Los diccionarios son especialmente útiles en situaciones en las que es necesario asociar un dato con un valor identificador. Por ejemplo, al almacenar datos de nombres de personas, es posible utilizar un diccionario en el que

los nombres estén asociados con los números de documento de cada individuo. De esta manera, al proporcionar el número de documento al diccionario, se obtendría el nombre correspondiente asociado a ese documento.

Si bien es posible lograr una funcionalidad similar utilizando arreglos y utilizando el valor clave como índice del arreglo, en muchos casos el valor clave no puede ser equiparado a un índice de arreglo. Por ejemplo, si se requiere almacenar datos de personas y utilizar el número de documento como índice de un arreglo, se enfrentaría a la impracticabilidad de crear un arreglo con 100 millones de elementos para cubrir todas las posibilidades de números de documento de 8 dígitos. Los diccionarios, en cambio, permiten utilizar cualquier tipo de dato disponible en Python como clave, ya sean valores numéricos, cadenas u otros tipos de datos.

La flexibilidad de los diccionarios radica en su capacidad para establecer relaciones directas entre datos y valores identificadores, independientemente de su tipo o dominio. Esto los convierte en una herramienta poderosa para abordar una amplia gama de situaciones donde es necesario asociar y recuperar datos de manera eficiente.

4.2.2. Operaciones

Creación

La sintaxis utilizada para crear diccionarios es bastante similar a la de los conjuntos. Existen tres mecanismos principales para crear un diccionario.

El primero de ellos es utilizando la función `dict()`, la cual inicializa un diccionario vacío al que se le pueden ir agregando pares clave-valor.

Otra forma de crear un diccionario es mediante la enumeración explícita de sus datos. Se utiliza un par de llaves ({}) y se enumeran los valores separados por comas. Sin embargo, a diferencia de los conjuntos, en los diccionarios se debe establecer un par clave-valor para cada valor enumerado. Esto se realiza utilizando dos puntos (:) para asociar la clave con su respectivo valor.

Por último, se puede utilizar la comprensión de diccionarios para crear diccionarios de manera concisa. La sintaxis es similar a la comprensión de conjuntos, utilizando un par de llaves que encierra una expresión de comprensión. En este caso, se establece el par clave-valor, donde tanto la clave como los valores pueden depender de los valores iterados en un ciclo for, utilizando dos puntos (:) para asociarla con el valor correspondiente.

Acceso

Una vez que se ha creado un diccionario, es posible acceder a sus valores utilizando el operador de corchetes ([]), que permite el acceso indexado similar al de los arreglos. En este caso, se indica la clave correspondiente al valor que se desea obtener. A diferencia de los arreglos, en los diccionarios la clave puede ser de cualquier tipo válido en Python. Por lo tanto, se puede utilizar el nombre del diccionario seguido del operador de acceso indexado, y dentro de éste se especifica la clave que se busca. El acceso indexado devuelve en tiempo constante el valor asociado con la clave. Si la clave no existe en el diccionario, se devuelve el valor vacío `None`.

Otra forma de acceder a los valores almacenados en el diccionario es utilizando el método `get()`. Este método recibe como parámetro la clave que se busca y, en caso de que la clave no se encuentre en el diccionario, devuelve None por defecto. Sin embargo, get() acepta un segundo parámetro opcional que permite especificar un valor por defecto. De esta manera, se puede acceder al valor asociado a la clave sin necesidad de utilizar una instrucción condicional, y se obtendrá el valor por defecto si la clave no existe en el diccionario.

Por último, existe un método llamado `setdefault()`, que recibe una clave y un valor por defecto como parámetros. Si la clave existe en el diccionario, el método devuelve el valor asociado a esa clave. En caso contrario, crea una nueva entrada en el diccionario asociando la clave con el valor por defecto especificado en el segundo parámetro. Este método es útil para evitar la verificación de existencia de la clave y asignar un valor predeterminado si no se encuentra en el diccionario.

Para poder conocer la longitud de un diccionario se dispone de la instrucción `len` que funciona de la misma manera que con todas las colecciones e iterables.

Inserción

Para agregar nuevos elementos o modificar elementos existentes en un diccionario, se utiliza el operador de acceso indexado de manera similar a como se haría con un arreglo. Es decir, se utiliza el nombre del diccionario seguido del símbolo de corchetes con la clave deseada en su interior, y luego se utiliza el operador de asignación (=) para asignar un nuevo valor a esa clave.

Si la clave ya existe en el diccionario, el valor asociado a esa clave se actualizará con el nuevo valor asignado. En caso de que la clave no exista previamente en el diccionario, se creará una nueva entrada con la clave y el

valor especificados. De esta manera, se puede agregar o modificar elementos en el diccionario de forma dinámica y flexible.

Borrado

Para eliminar elementos de un diccionario, existen dos métodos principales. El primero es utilizar la instrucción ‘del‘, seguida del nombre del diccionario y entre corchetes la clave que se desea borrar. Esta instrucción simplemente elimina la clave y el valor asociado a ella.

El segundo método es el uso del método ‘pop()‘. Este método recibe como argumento la clave que se desea eliminar y devuelve el valor asociado a esa clave. Además, elimina la clave y su valor del diccionario. Si se proporciona un segundo argumento opcional al método ‘pop()‘, se especifica un valor por defecto que será retornado si la clave no existe en el diccionario. Esto evita que se produzca un error en caso de que la clave no exista.

Otra opción es utilizar el método ‘clear()‘, el cual borra todos los pares clave-valor del diccionario, dejándolo vacío como si estuviera recién creado. Este método no requiere ningún argumento.

Finalmente, la instrucción ‘del‘ también puede utilizarse para eliminar todo el diccionario por completo, incluyendo sus valores contenidos, lo cual implica eliminar también la variable del diccionario en sí.

Estas opciones permiten la manipulación y eliminación de elementos en un diccionario de acuerdo a las necesidades específicas del programa.

Recorrido

Los diccionarios ofrecen tres formas de ser recorridos. El método keys() retorna un conjunto con todas las claves, el cual puede ser recorrido y manipulado con operaciones de conjuntos. Por otro lado el método values() retorna una lista con todos los valores. Corresponde que sea una lista porque si buen las claves no pueden repetirse, los valores sí pueden hacerlo.

Tambien pueden recorrerse los pares ordenados (clave, valor) con un ciclo for. Pero en este caso el ciclo posee dos variables que se asignan en cada vuelta. Para ello se dispone del método items() que retorna una secuencia de tuplas. En cada vuelta del for puede asignarse la tupla o directamente desempaquetar la misma en dos variables. En el ejemplo siguiente se las desempaqueta en las variables numero y nombre:

```
diccionario = {1: "Lunes", 2: "Martes", 3: "Miércoles"}

for numero, nombre in diccionario.items():
    print(f"El día {numero} se llama {nombre}")
```